# 1 INTRODUCTION

*Jealous stepmother and sisters; magical aid by a beast; a marriage won by gifts magically provided; a bird revealing a secret; a recognition by aid of a ring; or show; or what not; a dénouement of punishment; a happy marriage - all those things, which in sequence, make up Cinderella, may and do occur in an incalculable number of other combinations.*

— MR. Cox ***1893***, *Cinderella: Three hundred and forty-five variants [**?** ]*

The first web browser, Nexus, made its appearance in 1990 [**?** ]. At its inception, web browsing consisted solely of retrieving and displaying small, static textual web pages. In simpler terms, users could only read page content without any interactive components. However, the escalating computing power of devices, the proliferation of the internet, the valuation of internet based companies and the demand for more engaging user experiences birthed the concept of executing code in conjunction with web pages. In 1995, the Netscape browser revolutionized this concept by introducing JavaScript [**?** ], a language that allowed code execution on the client-side. Interactive web content immediately highlighted benefits: unlike classical native software, web applications do not require installation, are always up-to-date, and are accessible from any device with a web browser. Significantly, since the advent of Netscape, all browsers have offered JavaScript support. In the present day, the majority of web pages incorporate not only HTML but also JavaScript code, which is executed on client computers. Over the past several decades, web browsers have metamorphosed into JavaScript language virtual machines. They have evolved into intricate systems capable of running comprehensive applications, such as video and audio players, animation creators, and PDF document renderers, like the one displaying this document.

JavaScript is presently the most widely utilized scripting language in all contemporary web browsers [**?** ]. However, it is not without limitations due to the inherent characteristics of the language. For instance, each JavaScript engine necessitates the parsing and recompiling of the JavaScript code, resulting in substantial overhead. In fact, just parsing and compiling JavaScript code consume

[0] Compilation probe time 2023/10/30 10:07:58

the majority of the load times of websites [? ]. In addition to performance limitations, JavaScript also has security concerns [? ]. A notable example of this is the lack of memory isolation in JavaScript, which allows extraction of information from other processes [? ]. These issues led the Web Consortium (W3C) to standardize a bytecode for the web environment in 2015, which is the WebAssembly (Wasm) language. Hence, WebAssembly became the fourth official language for the web.

WebAssembly was designed with a focus on speed, portability, self-containment, and security [? ]. It allows for all programs to be compiled ahead-of-time from source languages like C/C++ and Rust. Third-party compilers create WebAssembly binaries, potentially including optimizations, as in the case of LLVM. The Wasm language defines its Instruction Set Architecture as an abstraction, similar to machine code instructions but independent of CPU architectures [? ]. This design allows web browsers to compile quickly to target architectures through a fast, one-to-one translation process.

**TODO** One paragraph here about the format of Wasm

The versatility of WebAssembly extends not just to web browsers, but to backend scenarios as well. Research has demonstrated the advantages of utilizing WebAssembly as an intermediary layer, noting improved startup times and more efficient memory usage when compared to containerization and virtualization [? ? ]. In response to these findings, the Bytecode Alliance introduced the WebAssembly System Interface (WASI) in 2019 [? ? ]. WASI facilitated the execution of WebAssembly with a POSIX system interface protocol, thus enabling the direct execution of Wasm in the operating system.

## 1.1 WebAssembly security

**TODO** Expand here, maybe separate in two paragraphs

WebAssembly is praised for its security, especially for its design that prevents programs from accessing data beyond their own memory. However, there has been less focus on potential vulnerabilities and attacks within WebAssembly's own memory [? ]. In addition, WebAssembly binaries may be inherently vulnerable due to flaws in their source code [? ]. There are also significant risks from side-channel attacks, as demonstrated by various researchers [? ? ? ]. These vulnerabilities are not limited to the web browser environment, as WebAssembly is also used in the backend.

## 1.2 Software Monoculture

Web browsers and JavaScript have evolved significantly in the past thirty years, leading to numerous implementations. Yet, only Firefox, Chrome, Safari, and Edge are commonly used on devices. This situation reflects a software

monoculture problem wherein a single flaw could impact multiple applications. The concept of monoculture is borrowed from biology and symbolizes an ecosystem at risk of extinction due to shared vulnerabilities and lack of diversity. Currently, web pages including WebAssembly binaries are centrally served from main servers. Thus, this monoculture issue is also applicable to the WebAssembly code served to web browsers. Despite its design for secure execution and sandboxing, Wasm is not immune to vulnerabilities. Therefore, sharing Wasm code through web browsers could also share its vulnerabilities.

The software monoculture problem exacerbates when considering the edge-cloud computing platforms and their adoption of WebAssembly to provide services. Specifically, in addition to browser clients, thousands of edge devices running WebAssembly as backend services could be affected by shared vulnerabilities. This scenario suggests that if one node in an edge network is vulnerable, all the others would be vulnerable in the exact same way since the same binary is replicated on each node. In simpler terms, the same attacker payload could compromise all edge nodes simultaneously, meaning that a single distributed Wasm binary could trigger a worldwide catastrophe.

## 1.3 WebAssembly malware

WebAssembly is often used in browsers for computation-intensive activities, including gaming and image processing, but it has also been exploited by malicious actors for cryptojacking [**?** ]. The popularity of WebAssembly for cryptojacking stems from its capacity for executing high volumes of hash functions, a task that demands substantial computing resources and is faster than JavaScript. Besides, WebAssembly code's poor readability makes it a convenient tool for obfuscating harmful code. Cryptojacking via WebAssembly often involves a malicious JavaScript+WebAssembly payload that secretly executes on the victim's browser [**?** ]. It's commonly found on websites offering illegal downloads or adult content, using cryptojacking as a means to generate passive income. Because it is hard to detect and remove, cryptojacking can persist on a victim's computer, continuously using resources and generating income for the attacker. This profitable malware does not require exploiting vulnerabilities or stealing credentials.

Several techniques employ static analysis, dynamic analysis and even state-of-the-art machine learning methods to detect WebAssembly crytomalware [**? ? ? ? ? ?** ]. However, obfuscation studies have revealed weaknesses in these methods, indicating a largely unexplored threat to malware detection accuracy in Wasm. Remarkably, these studies do not consider the existence of obfuscation tools. Bahnsali et al. have shown that cryptomining algorithms can evade these techniques through obfuscation [**?** ].

## 1.4 Problem statements

According to the discussion above, we identify three key problems to be addressed.

**Ps1 WebAssembly security**: WebAssembly ecosystem and binaries are vulnerable to attacks, specially side-channel attacks. Existing WebAssembly research mostly reacts to existing vulnerabilities, leaving the potential for unidentified attacks. Besides, current defenses are limited to specific attacks or require the alteration of runtimes.

**Ps2 Software monoculture**: Identical WebAssembly binaries are deployed on multiple nodes and browsers. Deployment systems, including web browsers, might be also identical. Such a situation presents a potential threat to the entire ecosystem due to shared vulnerabilities.

**Ps3 WebAssembly malware**: WebAssembly malware is a threat, while currently implemented defenses are not enough mostly due to the misconception of missing obfuscators.

## 1.5 Contributions in Software Diversification

This dissertation introduces tools, strategies, and methodologies designed to address the previously enunciated problem statements via Software Diversification. Software Diversification is a security-focused process that involves identifying, developing, and deploying program variants of a given original program [**?** ]. Pioneers in this field, Cohen et al. [**?** ] and Forrest et al. [**?** ], proposed enhancing software diversity through code transformations. Their proposal suggested creating program variants while maintaining their functionalities to mitigate potential vulnerabilities. Subsequently, transformations aimed at reducing the predictability of programs' observable behavior have been suggested. For instance, some researchers proposed diversifying the control flow of programs [**?** ], their instruction set [**?** ], or the system calls they utilize [**?** ]. A combination of these transformations can yield less predictable and therefore more secure variants.

While previous studies on software diversification have illustrated the removal of vulnerabilities, it can consistently serve as a preemptive solution in every instance. Despite the extensive research on software diversification, its application to WebAssembly remains largely unexplored. Firstly, software diversification could enhance the capabilities of established WebAssembly analysis tools by including diversified program variants, thus complicating the task for attackers to exploit any overlooked vulnerabilities. Generated as a proactive security measure, these diversified variants could mimic a wider range of real-world conditions, consequently increasing the accuracy of WebAssembly analysis tools, including WebAssembly malware detectors. Secondly, we observed that current solutions

to reduce side-channel attacks on WebAssembly binaries are either limited to specific attacks or require the alteration of runtimes. Software diversification could address yet-to-be-discovered vulnerabilities on WebAssembly binaries by creating diversified variants in a manner that is independent of the platform. In the realm of software diversification, we present the following contributions.

**C1 Experimental contribution:** For each proposed technique we provide an artifact implementation and conduct experiments to assess its capabilities. The artifacts are publicly available. The protocols and results of assessing the artifacts provide guidance for future research.

**C2 Theoretical contribution:** We propose a theoretical foundation in order to generate and improve Software Diversification for WebAssembly. We provide a formal definition of WebAssembly program variants and their diversity. We also provide a formal definition of WebAssembly program diversity generation.

**C3 Diversity generation:** We generate WebAssembly program variants. The variants are semantically equivalent to the original program, yet behaviorally diverse.

**C4 Defensive Diversification:** We assess how generated WebAssembly program variants could be used for defensive purposes. We provide empirical insights about practical usage of the generated variants in preventing side-channel attacks.

**C5 Offensive Diversification:** We evaluate the potential for using generated WebAssembly program variants for offensive purposes, while also enhancing security systems. Our research includes experiments where we test the resilience of WebAssembly analysis tools against these generated variants. Furthermore, we offer insights into which types of program variants practitioners should prioritize to improve WebAssembly analysis tools.

## 1.6 Summary of research papers

This compilation thesis comprises the following research papers. In Table 1.1 we map the contributions to our research papers.

**P1**: **CROW: Code randomization for WebAssembly bytecode.**
**Javier Cabrera-Arteaga**, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus

| Contribution | Research papers | | | |
|---|---|---|---|---|
| | P1 | P2 | P3 | P4 |
| C1 Experimental contribution | ✓ | ✓ | ✓ | ✓ |
| C2 Theoretical contribution | ✓ | | ✓ | |
| C3 Diversity generation | ✓ | ✓ | ✓ | ✓ |
| C4 Defensive diversification | ✓ | ✓ | ✓ | |
| C5 Offensive diversification | | | | ✓ |

***Table 1.1:*** *Mapping between contributions and research papers .*

*Measurements, Attacks, and Defenses for the Web (MADWeb 2021), 12 pages*
`https://doi.org/10.14722/madweb.2021.23004`

**Summary:** In this paper, we introduce the first entirely automated workflow for diversifying WebAssembly binaries. We present CROW, an open-source tool that implements software diversification through enumerative synthesis. We assess the capabilities of CROW and examine its application on real-world, security-sensitive programs. In general, CROW can create statically diverse variants. Furthermore, we illustrate that the generated variants exhibit different behaviors at runtime.

**P2**: **Multivariant execution at the Edge.**
**Javier Cabrera-Arteaga**,Pierre Laperdrix, Martin Monperrus, Benoit Baudry
*Moving Target Defense (MTD 2022), 12 pages*
`https://dl.acm.org/doi/abs/10.1145/3560828.3564007` **Summary:**

In this paper, we synthesize functionally equivalent variants of a deployed edge service. These variants are encapsulated into a single multivariant WebAssembly binary, When executing the service endpoint, a random variant is selected each time a function is invoked. Execution of these multivariant binaries occurs on the global edge platform provided by Fastly, as part of a research collaboration. We demonstrate that these multivariant binaries present a diverse range of execution traces throughout the entire edge platform, distributed worldwide, effectively creating a moving target defense.

**P3**: **Wasm-mutate: Fast and efficient software diversification for WebAssembly.**
**Javier Cabrera-Arteaga**, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry
*Under review, 17 pages*
`https://arxiv.org/pdf/2309.07638.pdf`

**Summary:** This paper introduces WASM-MUTATE, a compiler-agnostic

WebAssembly diversification engine. The engine is designed to swiftly generate semantically equivalent yet behaviorally diverse WebAssembly variants by leveraging an e-graph. We show that WASM-MUTATE can generate tens of thousands of unique WebAssembly variants in mere minutes. Importantly, WASM-MUTATE can safeguard WebAssembly binaries from timing side-channel attacks, such as Spectre.

**P4**: **WebAssembly Diversification for Malware evasion.**
**Javier Cabrera-Arteaga**,Tim Toady, Martin Monperrus, Benoit Baudry
*Computers & Security, Volume 131, 2023, 17 pages*
**Summary:** WebAssembly, while enhancing rich applications in browsers, also proves efficient in developing cryptojacking malware. Protective measures against cryptomalware have not factored in the potential use of evasion techniques by attackers. This paper delves into the potential of automatic binary diversification in aiding WebAssembly cryptojacking detectors' evasion. We provide proof that our diversification tools can generate variants of WebAssembly cryptojacking that successfully evade VirusTotal and MINOS. We further demonstrate that these generated variants introduce minimal performance overhead, thus verifying binary diversification as an effective evasion technique.

## Thesis layout

This dissertation comprises two parts as a compilation thesis. Part one summarises the research papers included within, which is partially rooted in the author's licentiate thesis [**?** ]. Chapter 2 offers a background on WebAssembly and the latest advancements in Software Diversification. Chapter 3 delves into our technical contributions. Chapter 4 exhibits two use cases applying our technical contributions. Chapter 5 concludes the thesis and outlines future research directions. The second part of this thesis incorporates all the papers discussed in part one.